Safe and Automated State Transfer for Secure and Reliable Live Update

Cristiano Giuffrida
Department of Computer Science
Vrije Universiteit, Amsterdam
giuffrida@cs.vu.nl

Andrew S. Tanenbaum
Department of Computer Science
Vrije Universiteit, Amsterdam
ast@cs.vu.nl

***Abstract*—Traditional live update systems offer little or no automated support for state transfer between two different program versions with changes in the program state. In this paper, we report our efforts to build a safe and automated state transfer framework for C programs that requires a minimal number of program state annotations and handles common structural state changes with no programmer assistance. To handle more complex state transformations, the framework includes a number of extension mechanisms designed to minimize the overall programming effort. Our experience with real-world programs suggests that our framework can handle all the standard C idioms and support safe and automated state transfer for complex state changes. We believe our approach is effective in several update scenarios and significantly raises the bar on the security and reliability of live update.**

I. Introduction

In recent years, the increasing demand for high availability has fostered much research on live update, due to its ability to apply software updates on a running system with no service interruption. Traditional live update systems glue code changes directly into a running program and employ some indirection mechanism (compiler-based techniques [1], [2], language support [3], or binary rewriting [4]–[7]) to redirect execution to the new version. State changes are handled by directly transforming the address space of the original program, typically using object reallocation [2], [3], [6], type wrapping [1] or shadow data structures [5], [7].

While effective for small security patches [4], [7], in-place updates scale poorly with the complexity of the update. This is particularly evident when state changes come into play. First, state transfer is delegated entirely to the programmer, who is forced to deal with a cross-version execution environment and reason about the correctness of state transfer code in all the possible update states. In particular, existing solutions offer no support to automatically locate and readjust pointers to updated data structures. If manual inspection fails to identify these cases, the update can silently generate pointers to stale types or objects, introducing subtle logical errors or, worse, security vulnerabilities. Second, assuming a fixed memory layout complicates or hampers nontrivial state transformations and compiler optimizations, increases version management complexity, and encourages memory leakage (e.g., no reclaiming of stale static portions of the state). Third, state transfer is neither checked for correctness nor isolated in a sandboxed environment, thus making the state transfer code inherently trusted. Finally, no generic support is available for bug-fix updates and state transfer code dealing with a tainted state (e.g., memory leakage or state corruption). All these observations are particularly critical for the security and reliability of live update.

We believe process-level live updates, first proposed by Gupta [8], have the potential to solve many of the recurring problems in existing solutions, with the state transferred between two processes running different program versions. Gupta's work, however, still makes assumptions on the memory layout of the processes and delegates state transfer entirely to the programmer. Ekiden [9] is a recent step forward in this direction, relaxing assumptions on the memory layout but still requiring the programmer to manually write serializers and deserializers for the state to transfer.

This paper presents a safe and automated state transfer framework for C programs, which seeks to minimize the programmer involvement in terms of: (i) state annotations and (ii) custom state transfer code to handle pointers and complex state transformations. Our framework operates at the process level, to sandbox the execution of the (untrusted) state transfer code in the new process and perform safe rollback in case of runtime errors. In addition, our framework supports automated *state checking* and *tainted state management*, including automatic reclaiming of memory leakages. We also support all the standard C idioms, which involves several challenges including *naming* and *pointer ambiguity*, as we explain later. We believe our framework is an important step forward over existing solutions and sets new standards for the security and reliability of live update.

II. State Instrumentation

Our state transfer framework requires detailed metadata information on all the possible objects that make up the program state. To record all the necessary metadata and expose them to the runtime, we transform the original program during the build process. To this end, our framework includes a specialized compiler driver, designed to integrate seamlessly with existing build systems. The transformation is implemented as a link-time pass using the LLVM compiler framework [10]. Our transformation pass covers the entirety of the program state and the static libraries (shared libraries are discussed later). To instrument the state in a fine-grained manner and automate the state transfer process, we first record metadata on the types (e.g., arrays or `structs`) used in the program. This allows us to correctly introspect the

state at runtime and simplify reasoning about type transformations. We only record types for all the state objects that play a role in the state transfer process.

Global variables. Global variables are an important part of the program state. For this reason, our transformation pass records metadata to describe and locate all the global and static variables (including constants).

Functions. While functions are automatically created in the new process and need not be transferred, they play an important role in function pointer transfer. Hence, we record metadata for all the functions whose address is taken.

String constants. Similar to functions, string constants are immutable objects and need not be normally transferred. To transfer string pointers correctly, however, we need to record metadata for all the string constants available.

Dynamically allocated objects. Dynamically allocated objects are a major source of state information and normally need to be transferred to the new version. To correctly introspect dynamically allocated objects or arrays, we need to determine the appropriate type of each memory allocation site in the code and create metadata for the corresponding allocated objects at runtime. Our transformation pass can automatically identify the correct type for all the standard POSIX memory allocators and custom allocators that use simple allocation wrappers. For more advanced custom allocation schemes, e.g., region-based memory allocators, the framework needs to be explicitly instructed to locate the proper allocation wrappers correctly.

Local variables. Although local variables do not usually carry any relevant state information, it is common for real C programs to store portions of the state in long-lived stack regions. Since we want our framework to work with different live update models [11] and update point placement strategies [1], we put no restriction on the number of local variables that can be promoted to state objects. By default, our framework creates metadata only for the local variables in `main`, but it is possible to specify a custom set of functions to instrument. Our pass can also be configured to automatically identify the functions to instrument from the set of update points, using static call-graph analysis. Our focus is on long-running event-driven programs, where update points are typically placed in the deepest long-lived function (e.g., at the top of the event loop), but other strategies are possible for complex multithreaded programs. The only constraint we make on update points is a well-defined mapping in each program version. This is to simplify control flow transfer and avoid unnecessary ambiguity in local variable transfer.

Shared libraries. Shared libraries can be instrumented similar to the original program and the resulting metadata made available at dynamic linking/loading time. This strategy, however, has a serious impact on deployability. For this reason, our framework also supports dynamic metadata generation for uninstrumented ELF libraries. While the type information is lost and fine-grained introspection is no longer possible, this strategy is important for a number of reasons. First, this is crucial for the precision of our dynamic pointer analysis, detailed later. Second, assuming no changes in the shared libraries after the update and no internal references to application objects (e.g., library pointers to application callbacks), automating library state transfer is possible by remapping the libraries at the same location in the new version and simply copying over the data regions. Finally, the programmer can use the provided metadata information to handle the more complicated cases, e.g., by reinitializing the library state in the new version.

III. State Objects Pairing

Our framework defines a uniform naming scheme to pair state objects in the old version to their counterparts in the new version and transfer the data at the process level correctly. The naming scheme is designed to be *stable* (i.e., unchanged objects are always assigned the same name in different versions), *unambiguous* (i.e., name clashing is structurally prevented), and *conservative* (i.e., the default pairing strategy only pairs objects with a well-defined mapping in the two versions). For this purpose, our scheme uses both names and contextual information extracted from the source code via static analysis. This approach provides a well-defined default pairing strategy, which is easy to understand and effective to handle common state changes automatically. To handle the ambiguous cases (e.g., variable renaming), the programmer can override the default pairing strategy using state transfer extensions. We are also working on a patch analysis tool to safely automate state transfer for some of the more complex transformations.

Our default pairing strategy pairs global variables and functions by name. Static global variables and static functions are paired by name and by compilation unit to unambiguously identify their context and prevent name clashing. Static and nonstatic local variables, in turn, are paired by name and by function of declaration. Note that our naming scheme (and the state transfer framework in general) is resilient to compiler optimizations like inlining, since the naming information is gathered from the original source code before any optimizations are made.

The default pairing strategy for string constants considers two cases: string initializers (i.e., constants commonly used to initialize local or global pointers) and regular strings. By default, string initializers follow the pairing strategy of their parent pointer (i.e., the string initializer of a pointer in the old version is paired with the string initializer of the pointer counterpart in the new version). This strategy allows us to automate state transfer of string pointers even in face of initializer changes. For regular strings, pairing is simply done basing on the content of the string, by default.

Pairing dynamically allocated objects is more challenging, because they have no preexisting counterpart in the new version. Our approach is to use static analysis to determine the allocation context of each dynamically allocated object and define a default pairing strategy accordingly. For each allocation site, we record metadata on the parent function,

the pointer used to allocate memory, and a counter to prevent name clashing in the ambiguous cases. By default, dynamically allocated objects are paired (and automatically reallocated in the new version when needed) only when an exact match is found between the allocation sites in the two versions. This conservative strategy reflects the intuition that unpaired allocation sites translate to either dynamically allocated objects no longer used in the new version or complex changes that require programmer intervention.

IV. The State Transfer Process

The state transfer process begins when the old version reaches a valid update point for a given available update (as defined by the particular update model used). At that point, a helper routine creates a fresh process image for the new version and blocks waiting for state transfer events. The new process starts executing and receives all the update-specific information via command-line arguments. Control is given to our framework right after registering state transfer extensions (if any). Note that both the framework and all the registered extensions run sandboxed in the new version.

When state transfer completes successfully, the new version jumps to the update point counterpart (with a well-defined mapping with the old update point) to resume execution. The old version is immediately cleaned up. Conversely, when an error occurs during state transfer, the update is promptly rollbacked. The new version is cleaned up and control is given back to the old version to resume execution as though the update never occurred. All the actions are coordinated by an external monitor process, which arbitrates control transfer and traces the execution of the new process to detect runtime errors such as crashes, abnormal terminations, and infinite loops.

To transfer the state from the old version to the new version, our framework supports a pluggable IPC (Inter-Process Communication) mechanism to copy memory regions from the old process to the new one. We are experimenting with both *cooperative* and *uncooperative* IPC mechanisms. Uncooperative mechanisms (e.g., tracing) are typically slower but do not require the old version to be involved in the state transfer process. Cooperative mechanisms (e.g., UNIX domain sockets) require the old version to run an IPC server to process memory copying requests issued by the new version. Although potentially faster, this approach requires some untrusted state transfer code running on the old version. While typically small in size, this code can potentially lower the safety guarantees provided by our framework. We are carefully evaluating the different tradeoffs in our ongoing work. Figure 1 depicts the resulting state transfer process.

Our design breaks down the state transfer process into a number of different phases. State transfer extensions can register handlers (i.e., callbacks) and override the default behavior at any stage during the process.

Metadata transfer. In the first phase, the state transfer framework transfers all the state metadata from the old version to the (local) address space of the new version.

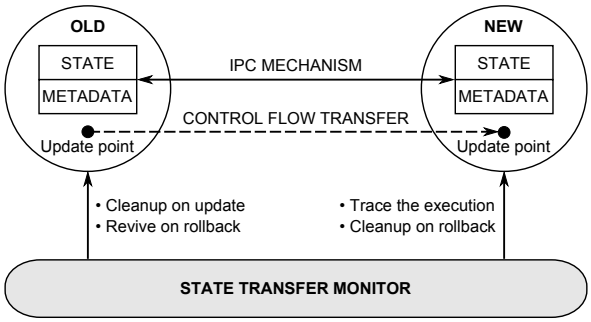

Figure 1: The state transfer process.

The state metadata are generated by our instrumentation and include all the necessary information to locate and describe every state object in the old version. The metadata are based on a fixed layout and placed at a known location, allowing our framework to easily automate metadata transfer. With both the old and the new metadata available locally, the framework and the state transfer extensions can introspect the state of the two versions in the subsequent phases.

State mapping. The outcome of this phase is a well-defined mapping between the state objects in the old version and their counterparts in the new version. When no state transfer extension is registered, the framework automatically follows the default pairing strategy described in Section III and schedules all the state objects for transfer. Extensions are allowed to override the default behavior by scheduling only selected objects for transfer or adding new pairing rules. The only constraint enforced is that each object in the old version must be paired to at most one counterpart in the new version. Extensions are also allowed to pair state objects of different nature. For example, a long-lived local variable can be paired to a global variable in the new version, or a statically allocated buffer can be paired to a dynamically allocated one in the new version. This is especially beneficial for updates that originate from code refactoring.

State pre checking. Before transferring any data to the new version, our framework checks the consistency of the old state. This is important to ensure predictability and allow for safe state transfer in the case of a tainted state. Our default checking strategy is fully automated, but extensions can be registered to enhance its accuracy or override the default behavior when tainted state objects are found (abnormal termination). Our current checking strategy uses dynamic pointer analysis to check the consistency of all the pointers (see Section V). In our ongoing work, we are extending the scope of our analysis and evaluating more complex state invariants to provide additional safety guarantees.

Data transfer. In this phase, the framework processes all the nonconstant state objects scheduled for transfer and orderly migrates their data to the new version. All the nontransferred state objects are simply left untouched, with their values normally initialized in the new version. To automate the transfer, the framework introspects individual state objects in the old version by walking their type recursively

and examining each inner state element found. Nonpointer elements are normally transferred by value, while pointer elements require a more careful transfer strategy, discussed in the next section. Each recursive step requires the framework to map the current state element to its counterpart in the new version. When two paired state objects have identical types, the mapping is straightforward. In case of type changes, locating the state element counterpart (if any) is more challenging. The default mapping strategy used in the framework follows a conservative approach, aimed at automating state transfer for many common structural changes, like: (i) primitive type transformations, (ii) array truncation or expansion, and (iii) addition, deletion, or reordering of `structs` members. As usual, the programmer can register extensions to override the default mapping strategy or implement custom state transfer policies at the element level. Extensions can also be used to transfer any *external* state correctly, for example state shared with other nonupdated processes or process-specific state not automatically transferred with the `fork() + exec()` paradigm.

State post checking. The last phase performs state checking on the new state. This is important to safeguard the new version against potential state corruption introduced by the untrusted state transfer code. Further, this allows us to automatically detect violating assumptions in state transfer. For example, consider a pointer in the old version legitimately pointing to some global variable. Assume both objects are transferred to the new version using an identity type mapping, i.e., the pointer counterpart points to the variable counterpart in the new version. Now consider the case in which the global variable does not have its address taken in the new version. Allowing the transfer would violate this assumption in the new version and bring the program to an invalid state when resuming execution. Our framework can instead detect this and other violations and prevent unwanted situations by immediately rolling back the update.

V. Pointer Transfer

A truly automated state transfer solution for C needs to carefully handle all the pointers in the program for the transfer to be predictable and safe. Failing to do so could result in arbitrary state corruption in the new version, undermining the security and reliability of live update. A corrupted state may lead to a number of undesirable situations, including silent data corruption and security vulnerabilities.

Unfortunately, many programming constructs allowed by C introduce different forms of *pointer ambiguity*, making pointer transfer particularly challenging. We have investigated all the cases of pointer ambiguity commonly found in well-formed C programs, including: casted pointers, interior pointers, pointers as integers, integers as pointers, unions with pointers, guard and dangling pointers. To handle these cases correctly, our framework leverages a combination of static and dynamic analysis, which requires programmer intervention (i.e., callbacks or annotations) only in the undecidable cases. While annotations may seem to hinder the automated nature of our framework, we see this as a feature rather than a limitation. Annotations can compensate for the manual effort to transfer portions of the state and readjust all the pointers to updated data structures. Failing to do so could lead to the security and reliability problems pointed out earlier. In our experience with real-world operating system code and server applications (e.g., `lighttpd`, `proftpd`, `exim`), region-based allocators and `unions` with pointers (discussed later) are frequently the only cases that require explicit programmer intervention. In our experience, this (normally one-time) effort is by far more bearable and realistic than manual code inspection to establish the validity of in-place updates with nontrivial state changes.

Pointers with a valid target. This is the common scenario of a pointer to a valid static or dynamic state object (target). To automate the transfer, our framework must locate the target counterpart in the new version. Since our state instrumentation provides accurate type information for all the state objects, the transfer strategy can be solely based on target-specific information with no assumption on the original pointer type. This is crucial for our design to be resilient to pointer casting and generic `void*` pointers. To map a particular target element to its counterpart, our framework resorts to the same mapping subsystem used earlier to remap regular state elements. This guarantees that state elements and their corresponding pointers are remapped accordingly, even in case of type changes. In addition, this significantly reduces the programming effort to write custom mapping extensions. Finally, this allows our framework to handle regular pointers and interior pointers (i.e., pointers into the middle of a data structure) uniformly.

When used in conjunction with pointer casting, however, interior pointers pose additional challenges. In particular, pointers to the beginning of a `struct` cannot be easily discriminated from pointers to its first element. This introduces ambiguity in case of type changes. To address this problem, our approach is to instrument the original code and add padding at the beginning of each non-`packed` `struct` (or `union`) type (i.e., type changes to packed structures may still need programmer assistance). When the target counterpart is found, the pointer is reinitialized accordingly and the target is automatically scheduled for transfer by default. This strategy preserves the structure of arbitrarily complex data structures during the transfer. When no counterpart is found, our framework requires the programmer to resolve the ambiguity and transfer the pointer correctly. Dynamically allocated objects that are not directly or indirectly referenced by any transferred pointer are not transferred or reallocated in the new version by default. This allows our framework to structurally prevent any memory leakage from propagating to the new version.

Integers as pointers. In real C programs, it is not uncommon for integers to be stored as pointers to indicate special values for particular pointer types (e.g., `MAP_FAILED` (`-1`) used by `mmap`). Our framework uses static analysis to mark all the pointers that can be assigned integer values

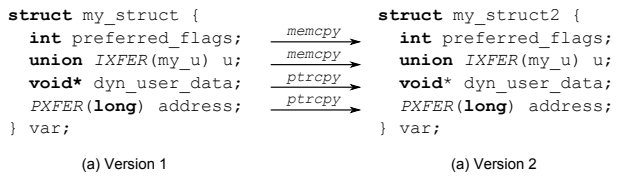

Figure 2: Type annotations for ambiguous transfer scenarios.

as *integer candidates*, while trying to determine their value set. When the value set cannot be accurately identified, we conservatively assume a value set of the form $[-P,\ P]$, where P is the architecture-specific page size. These ranges are reserved and commonly used to express special pointer values. At transfer time, the framework checks if the value of a given integer candidate matches any value in the value set. The value is simply copied to the new version when a match is found, otherwise the framework resorts to the default transfer strategy.

Pointers as integers. Occasionally, programmers find it convenient to store a memory address as an integer. This is not uncommon, for instance, in state objects part of custom memory management implementations. Unfortunately, this is a case of unresolvable ambiguity, which no automated static or dynamic analysis can easily settle in the general case. For this reason, our framework requires the programmer to explicitly annotate these pointers in the source code.

Unions with pointers. C allows `unions` to be written and read from using different layouts with no restriction. Consequently, their runtime type cannot be determined automatically in the general case and our framework generally requires programmer assistance to transfer unions correctly. Fortunately, this is only required in case of layout changes and for unions that contain pointers. To help the programmer deal with unions and other ambiguous cases, our framework supports variable and type annotations. An example of type annotations is presented in Figure 2, with the `IXFER` and `PXFER` annotations forcing the framework to `memcpy` the union `u` (without introspecting it) and perform pointer transfer of the integer `address`.

Guard pointers. Many programs use dedicated pointers to mark buffer boundaries. For example, off-by-N pointers may be used to mark the Nth element before or after the buffer. Our pointer analysis supports accurate detection and transfer of arbitrary guard pointers, as long as the original code is instrumented with a padding of N elements before and after each state buffer.

Dangling and invalid pointers. Dangling pointers (i.e., pointers to deallocated objects) are another case of interest. While uncommon in a stable update state, identifying such pointers is important to avoid unwanted behavior in the general case. It may also be beneficial to expose knowledge of dangling pointers to state transfer extensions that deal with a tainted state. Our framework marks as invalid all the pointers that cannot be recognized as one of the scenarios above and aborts the transfer by default. This ensures predictability and allows the framework to identify forms of corruption in a tainted state. Currently, we approximate identification of dangling pointers within the set of invalid pointers (using knowledge of known heap and stack regions), but we are planning to enhance the accuracy of our analysis in future work. Uninitialized pointers, in contrast, are prevented structurally by zeroing out newly allocated objects.

VI. Conclusion

In this paper, we presented a safe and automated state transfer framework to support secure and reliable live update. Our framework can handle pointers and all the other standard C idioms, providing an effective mechanism to perform complex state updates between two program versions. Our approach requires minimal programmer involvement while allowing arbitrary customizations. Unlike existing solutions, our design can support safe rollback in case of runtime errors, ensure a safe and predictable state transfer process, simplify tainted state management, and provide an effective programming model for state transfer extensions.

References

[1] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, "Practical dynamic software updating for C," *ACM SIGPLAN Notices*, vol. 41, no. 6, pp. 72–83, 2006.

[2] K. Makris and R. Bazzi, "Immediate multi-threaded dynamic software updates using stack reconstruction," in *Proc. of the USENIX Annual Tech. Conf.*, 2009, pp. 397–410.

[3] A. Baumann, J. Appavoo, R. W. Wisniewski, D. D. Silva, O. Krieger, and G. Heiser, "Reboots are for hardware: Challenges and solutions to updating an operating system on the fly," in *Proc. of the USENIX Annual Tech. Conf.*, 2007, pp. 1–14.

[4] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz, "OPUS: Online patches and updates for security," in *Proc. of the 14th USENIX Security Symp.*, vol. 14, 2005, pp. 19–19.

[5] K. Makris and K. D. Ryu, "Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels," in *Proc. of the Second ACM SIGOPS/EuroSys European Conf. on Computer Systems*, 2007, pp. 327–340.

[6] H. Chen, J. Yu, R. Chen, B. Zang, and P. Yew, "POLUS: A POwerful live updating system," in *Proc. of the 29th Int'l Conf. on Software Engineering*, 2007, pp. 271–281.

[7] J. Arnold and M. F. Kaashoek, "Ksplice: Automatic rebootless kernel updates," in *Proc. of the Fourth ACM European Conf. on Computer Systems*, 2009, pp. 187–198.

[8] D. Gupta and P. Jalote, "On line software version change using state transfer between processes," *Softw. Pract. and Exper.*, vol. 23, no. 9, pp. 949–964, 1993.

[9] C. M. Hayden, E. K. Smith, M. Hicks, and J. S. Foster, "State transfer for clear and efficient runtime updates," in *Proc. of the Third Int'l Workshop on Hot Topics in Software Upgrades*, 2011, pp. 179–184.

[10] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *Proc. of the Int'l Symp. on CGO*, 2004.

[11] C. Giuffrida and A. S. Tanenbaum, "Cooperative update: a new model for dependable live update," in *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades*, 2009, pp. 1–6.